Identifying Cascading Errors using Constraints in Dependency Parsing

Dominick Ng and **James R. Curran**

ə-lab, School of Information Technologies
University of Sydney
NSW 2006, Australia
{dominick.ng,james.r.curran}@sydney.edu.au

Abstract

Dependency parsers are usually evaluated on attachment accuracy. Whilst easily interpreted, the metric does not illustrate the cascading impact of errors, where the parser chooses an incorrect arc, and is subsequently forced to choose further incorrect arcs elsewhere in the parse.

We apply arc-level constraints to MSTparser and ZPar, enforcing the correct analysis of specific error classes, whilst otherwise continuing with decoding. We investigate the direct and indirect impact of applying constraints to the parser. Erroneous NP and punctuation attachments cause the most cascading errors, while incorrect PP and coordination attachments are frequent but less influential. Punctuation is especially challenging, as it has long been ignored in parsing, and serves a variety of disparate syntactic roles.

1 Introduction

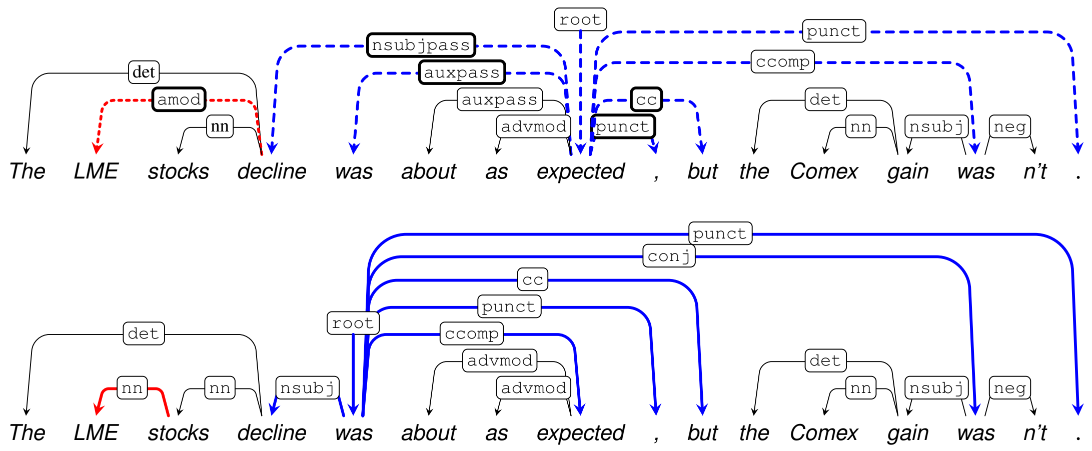
Dependency parsers are evaluated using word-level attachment accuracy. Whilst comparable across systems, this does not provide insight into why the parser makes certain errors, or whether certain misattachments are caused by other errors. For example, incorrectly identifying a modifier head may only introduce a single attachment error, while misplacing the root of a sentence will create substantially more errors elsewhere. In projective dependency parsing, erroneous arcs can also force the parser to select other incorrect arcs.

Kummerfeld et al. (2012) propose a static post-parsing analysis to categorise groups of bracket errors in constituency parsing into higher level error classes such as clause attachment. However, this cannot account for cascading changes resulting from repairing errors, or limitations which may prevent the parser from applying a repair. It is unclear whether the parser will apply the repair operation in its entirety, or if it will introduce other changes in response to the repairs.

We develop an evaluation procedure to evaluate the influence of each error class in dependency parsing without making assumptions about how the parser will behave. We define error classes based on dependency labels, and use the dependencies in each class as *arc constraints* specifying the correct head and label for particular words in each sentence. We adapt parsers to apply these constraints, whilst otherwise proceeding with decoding under their grammar and model. By evaluating performance with and without constraints, we can directly observe the cascading impact of each error class on each the parser.

We implement our procedure for the graph-based MSTparser (McDonald and Pereira, 2006) and the transition-based ZPar (Zhang and Clark, 2011) using basic Stanford dependencies over the OntoNotes 4.0 release of the WSJ Penn Treebank data. Our results show that erroneously attaching NPs, PPs, modifiers, and punctuation have the largest overall impact on UAS. Of those, NPs and punctuation have the most substantial cascading impact, indicating that these errors have the most effect on the remainder of the parse. Enforcing correct punctuation arcs has a particularly large impact on accuracy, even though most evaluation scripts ignore punctuation. We find that punctuation arcs are commonly misplaced by large distances in the final parse, crossing over and forcing other arcs to be incorrect in the process.

We will make our source code available, and



root
punct
nsubjpass
ccomp
auxpass
det
auxpass
cc
amod
nn
advmod
punct
det
nn
nsubj
neg
The LME stocks decline was about as expected , but the Comex gain was n't .
punct
conj
cc
punct
root
ccomp
det
advmod
nn
nn
nsubj
advmod
det
nn
nsubj
neg
The LME stocks decline was about as expected , but the Comex gain was n't .

Figure 1: MSTparser output (top) and the gold parse (bottom) for a WSJ 22 sentence. MSTparser produces two independent errors: an NP bracketing error (red, dotted), and an incorrect root (blue, dashed).

Parser	UAS	LAS	usent	lsent
MSTparser	91.3	87.5	41.3	26.1
ZPar	91.7	89.3	45.1	35.9

Table 1: Baseline UAS and LAS scores on Stanford dependencies over WSJ 22.

hope that our findings will drive efforts addressing the remaining dependency parsing errors.

2 Motivation

Table 1 summarises the performance of MSTparser and ZPar on Stanford dependencies over OntoNotes 4 WSJ 22. ZPar performs slightly better than MSTparser on UAS, and substantially better on LAS. However, these numbers do not show what types of errors are being made by each parser, what errors remain to be addressed, or hint at what underlying problems cause each error.

Figure 1 depicts a WSJ 22 sentence as parsed by MSTparser, and the gold parse. The UAS is 47.1%, with 8 of 17 arcs correct. By contrast, ZPar (parse not shown) scores 94.1%, with the sole attachment error being on *LME* (as with MSTparser). While there are nine incorrect arcs overall, MSTparser seems to have made only two underlying errors:

- *LME* attached to *decline* rather than *stocks* (NP internal). Correcting this repairs one error;
- *expected* being chosen as the root rather than *was*. Correcting the root and moving all attachments to it from the old root repairs the remaining eight errors.

Intuitively, it seems that the impact of the NP error is limited. By contrast, the root selection error has a substantial impact on the second half of the sentence, causing a misplaced subject, misattached punctuation, and incorrect coordination. These cascaded errors appear to be *caused by* the incorrect root.

What we do not know is whether these intuitions actually hold. Many dependency parsers, including MSTparser and ZPar, construct trees by repeatedly combining fragments together until a spanning analysis is found, using a small window of information to make each arc decision. An error in one part of the tree may have no influence on a different part of the tree. Alternatively, errors may exert long-range influence — particularly if there are higher-order features or algorithmic constraints such as projectivity over the tree. As parsing algorithms are complex, we wish to repair various error types in isolation without otherwise making assumptions regarding the subsequent actions of the parser.

3 Applying Constraints

We investigate how each parser behaves when certain errors in the tree are corrected. We force each parser to select the correct head and label for specific words, but otherwise allow it to construct its best parse. Given a set of *constraints*, each of which lists a word with the head and label to which it must be attached, we investigate two measures:

1. errors *directly corrected* by the constraints, called the *constrained accuracy impact*;

2. the *indirect impact of the constraints*, including errors *indirectly corrected*, and correct

arcs *indirectly destroyed*, together called the *cascaded accuracy impact*.

The constrained accuracy impact tells us how often the parser makes errors in the set of words covered by the constraints. The cascaded accuracy impact is less predictable, as it describes what effect the errors made over the constrained set of arcs have over the rest of the sentence. It is the *influence* of the set of constraints over the other attachments, which may be mediated through projectivity requirements, or changes in the context used for other parsing decisions.

The core of our procedure is adapting each parser to accept a set of constraints. Following Kummerfeld et al. (2012), we define meaningful error classes grouped with the operations that repair them. In dependency parsing, error classes are groups of Stanford dependency labels, rather than groups of node repair operations. The Stanford labels provide a rich distinction in NP internal structure, clauses, and modifiers, and map well to the error categories of Kummerfeld et al. (2012), allowing us to avoid excessive heuristics in the mapping process. Our technique can be applied to other dependency schemes such as LTH (Johansson and Nugues, 2007) by defining new mappings from labels to error types.

The difficulty of the mapping task depends on the intricacies of each formalism. The major challenge with LTH dependencies is the enormous skew towards the nominal modifier `NMOD` label. This label occurs 11,335 times in WSJ 22, more than twice as frequently as the next most frequent punctuation `P`. By contrast, the most common Stanford label is punctuation, at 4,731 occurrences. The `NMOD` label is split into many smaller, but more informative nominal labels in the Stanford scheme, making it better suited for our goal of error analysis.

The label grouping was performed with reference to the Stanford dependencies manual v2.04 (de Marneffe and Manning, 2008, updated 2012). For each error class, we generate a set of constraints over WSJ 22 for all words with a gold-standard label in the set associated with the class. Our types are defined as follows:

NP attachment: any label specifically attaching an NP, includes `appos`, `dobj`, `iobj`, `nsubj`, `nsubjpass`, `pobj`, and `xsubj`.

NP internal: any label marking nominal structure (not including adjectival modifiers), includes `abbrev`, `det`, `nn`, `number`, `poss`, `possessive`, and `predet`.

PP attachment: any label attaching a prepositional phrase, includes `prep`. Also includes `pcomp` if the POS of the word is `TO` or `IN`.

Clause attachment: any label attaching a clause, includes `advcl`, `ccomp`, `csubj`, `csubjpass`, `purpcl`, `rcmod`, and `xcomp`. Also includes `pcomp` if the POS of the word is not `TO` or `IN`.

Modifier attachment: any label attaching an adverbial or adjectival modifier, includes `advmod`, `amod`, `infmod`, `npadvmod`, `num`, `partmod`, `quantmod`, and `tmod`.

Coordination attachment: `conj`, `cc`, and `preconj`.

Root attachment: the `root` label.

Punctuation attachment: the `punct` label.

Other attachment: all other Stanford labels, specifically `acomp`, `attr`, `aux`, `auxpass`, `complm`, `cop`, `dep`, `expl`, `mark`, `mwe`, `neg`, `parataxis`, `prt`, `ref`, and `rel`.

For example, Root constraints specify sentence roots, while PP constraints specify heads of prepositional phrases.

One deficiency of our implementation is that we apply constraints to all arcs of a particular error type in each sentence, and do not isolate multiple instances of the same error class in a sentence. We do this since applying single constraints to a sentence at a time would require substantial modifications to the standard evaluation regime.

3.1 MSTparser implementation

MSTparser is a graph-based, second-order parser that uses Eisner (1996)'s algorithm for projective decoding (McDonald and Pereira, 2006).[1] Eisner's algorithm constructs and caches subtrees which span progressively larger sections of the sentence. These spans are marked either as *complete*, consisting of a head, a dependent, and all of the descendants of that head to one side, or *incomplete*, consisting of a head, a dependent, and an unfilled region where additional tokens may be attached. Dependencies are formed between the head and dependent in each complete span, while label assignment occurs as a separate process.

We enforce constraints by allowing complete spans to be formed only from constrained tokens

[1]As the variant of Stanford dependencies we use are projective, we did not use non-projective decoding.

to their correct heads with the correct labels. Any complete span between an incorrect head and the constrained token is forbidden. The algorithm is forced to choose the constrained spans as it builds the parse; these constraints have no impact on the parser's coverage as all possible head selections are considered.

3.2 ZPar implementation

ZPar is an arc-eager transition-based parser (Zhang and Clark, 2011) that uses an incremental process with a stack storing partial parse *states* (Nivre et al., 2004). Each state represents tokens that may accept further arcs. The tokens of a sentence are initially stored in a buffer, and at each point during parsing, the parser decides whether or not to create an arc between the *front* token of the buffer and the *top* token on the stack.

We apply constraints in a similar way to Nivre et al. (2014). Arc creation actions are factored on the dependency label to be assigned to the arc. ZPar scores each possible action using a perceptron model over features from the front of the buffer and the top of the stack (as well as some additional context features which refer to previously created states). The highest scoring actions and their resulting states are kept in a beam; during parsing, ZPar finds the optimal action for all items in the beam, and retains the highest scoring new states at each step.

We disallow any arc creation action that would create an arc that conflicts with any constraints. Due to the use of beam search, it is possible for all of the partial states containing the constrained arcs to be evicted from the beam if they score lower under the model than other states. When this happens, the parser will fail to find an analysis for the sentence, as no head will exist in the beam for the constrained token. We have deliberately chosen to not address this issue as any solution (e.g. increasing the beam size from its default of 64) would change the decisions of the parser and model.

We verified that our modifications were working correctly for both parsers by passing in zero constraints (checking that the output matched the baseline performance), and every possible constraint (checking that the output scored 100%).

4 Related Work

Kummerfeld et al. (2012) perform a comprehensive classification of constituency bracket errors and their cascading impact, and their work is philosophically similar to ours. They associate groups of bracket errors in the parse with abstract error classes, and identify the tree operations that repair these error types, such as the insertion, deletion, or substitution of nodes in the parse tree. The error classes in a particular parser's output are identified through a heuristic procedure that repeatedly applies the operation repairing the largest number of bracket errors. This approach differs from our methodology as it is a static post-process that assumes the parser would respond perfectly to each repair, when it is possible that the parser may not perform the repair in full, or even be incapable of constructing the repaired tree.

McDonald and Nivre (2011) perform an in-depth comparison of the graph-based MSTparser and transition-based MaltParser. However, MaltParser uses support vector machines to deterministically predict the next transition, rather than storing the most probable options in a beam like ZPar. Additionally, they do not focus on the cascading impact of errors, and instead concentrate on higher-level error classification (e.g. by POS tag, labels and dependency lengths) in lieu of examining how the parsers respond to forced corrections.

Nivre et al. (2014) describe several uses for arc-level constraints in transition-based parsing. However, these applications focus on improving parsing accuracy when constraints can be readily identified, e.g. imperatives at the beginning of a sentence are likely to be the root. We focus our constraints on evaluation, attempting to identify important sources of error in dependency parsers.

Our constraint-based approach shares similarities to oracle training and decoding methods, where an external source of truth is used to verify parser decisions. An oracle source of parser actions is a necessary component for training transition-based parsers (Nivre, 2009). Oracle decoding, where a system is forced to produce correct output if possible, can be used to assess its upper performance bounds (Ng and Curran, 2012).

Constraining the parser's internal search space is akin to an optimal pruning operation. Charniak and Johnson (2005) use a coarse-to-fine, iterative pruning approach for efficiently generating high-quality n-best parses for a discriminative reranker. Rush and Petrov (2012) use a similar coarse-to-fine algorithm with vine grammars (Eisner and Smith, 2005) to accelerate graph-based de-

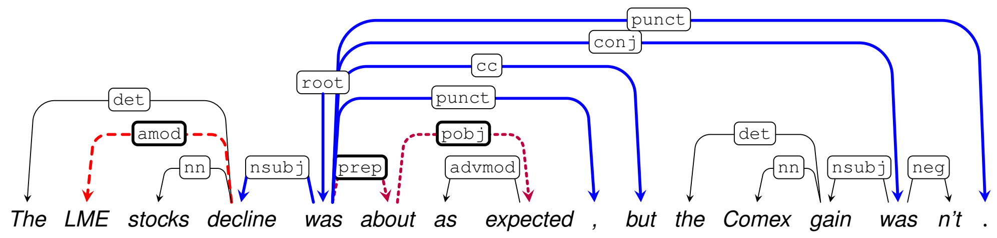

punct
conj
cc
root
punct
det
amod
pobj
nn
nsubj
prep
advmod
det
nn
nsubj
neg
The LME stocks decline was about as expected , but the Comex gain was n't .

Figure 2: MSTparser output for the sentence in Figure 1, where the `root` dependency is forced to its correct value. The incorrect noun phrase error is not affected by the constraint (dashed, red), six attachment errors are repaired (solid, blue), and two new errors are introduced (dotted, purple).

Constraints	✓	✗	Remaining Errors
None	8	9	see Figure 1
`nn`	9	8	All except *decline* → *LME*
`root`	14	3	*decline* → *LME* *about* → *expected* *was* → *about*
`punct`	14	3	*decline* → *LME* *about* → *expected* *was* → *about*
`ccomp`	16	1	*decline* → *LME*

Table 2: Correct and incorrect arcs, and the remaining errors after applying various sets of constraints to the sentence in Figure 1.

pendency parsing, achieving parsing speeds close to linear-time transition parsers despite encoding more complex features. Supertagging (Clark and Curran, 2007) and chart pruning (Zhang et al., 2010) have been used to constrain the search space of a CCG parser, and to remove unlikely or forbidden spans from repeated consideration. In our work, we use pruning not for parsing speed, but evaluation, and so we prune items based on gold-standard constraints rather than heuristics.

5 Evaluation

We use the training (sections 2-21) and development (section 22) data from the OntoNotes 4.0 release of the Penn Treebank WSJ data (Marcus et al., 1993), as supplied by the SANCL 2012 Shared Task on Parsing the Web (Petrov and McDonald, 2012). OntoNotes annotates enriched NP structure compared to the Penn Treebank (Weischedel et al., 2011), meaning that determining NP attachments is less trivial. We changed all marker tokens in the corpus (e.g. `-LRB-` and `-LCB-`) to their equivalent unescaped punctuation marks to ensure correct evaluation. The corpus has been converted to basic Stanford dependencies using the Stanford Parser v2.0,[2] and part-of-speech tagged using MXPOST (Ratnaparkhi, 1996). A model trained on WSJ sections 2-21 was used to tag the development set, and 10-fold jackknife training was used to tag the training data.

We implement a custom evaluation script to facilitate a straightforward comparative analysis between the unconstrained and constrained output. The script is based on and produces identical scores to `eval06.pl`, the official evaluation for the CoNLL-X Shared Task on Multilingual Dependency Parsing (Buchholz and Marsi, 2006). We ignore punctuation as defined by `eval06.pl` in our evaluation; experiments with constraints over punctuation tokens constrain those tokens in the parse, but ignore them during evaluation.

We run the modified parsers over WSJ 22 with and without each set of constraints. We examine the overall unlabeled and labeled attachment scores (UAS and LAS), as well as identifying the contribution to the overall UAS improvement from directly (constrained) and indirectly corrected arcs.

MSTparser uses coarse-grained tags and fine-grained POS tags in its features, both of which were provided by the CoNLL-X Shared Task. We approximate the coarse-grained POS tags by taking the first character of the MXPOST-assigned POS tag, a technique also used by Bansal et al. (2014)[3].

6 Results

Figure 2 and Table 2 show the impact of applying constraints on tokens with various labels to MSTparser for the sentence in Figure 1. Enforcing the gold `nn` arc between *decline* and *LME* repairs that noun phrase error, but does not affect any of the other errors. Conversely, enforcing the gold `root` arc does not affect the noun phrase error, but repairs nearly every other error in the parse. Unfortunately, the constrained `root` arc introduces

[2] `nlp.stanford.edu/software/lex-parser.shtml`

[3] Mohit Bansal, p.c.

Error class	cover	eff	eff %	disp	UAS	LAS	ΔUAS	Δc	Δu
Baseline	100.0	-	-	-	91.3	87.5	-	-	-
NP attachment	95.6	312	4.9	5.2	94.1	90.7	2.3	1.2	1.1
NP internal	98.2	206	3.2	2.8	92.6	89.2	1.1	0.8	0.3
Modifier attachment	96.8	321	7.9	3.8	93.4	90.3	1.7	1.2	0.5
PP attachment	98.3	378	13.1	4.3	93.2	89.5	1.7	1.4	0.3
Coordination attachment	97.7	238	16.0	5.8	92.9	89.5	1.3	0.9	0.4
Clause attachment	96.7	228	17.9	6.9	93.0	89.6	1.4	0.9	0.5
Root attachment	99.1	77	5.8	9.3	92.2	88.3	0.8	0.3	0.5
Punctuation attachment	93.2	469	14.2	7.4	93.9	89.9	1.8	0.1	1.7
Other attachment	94.3	210	7.0	6.1	93.5	90.8	1.4	0.8	0.6
All attachments	98.5	2912	9.3	5.8	100.0	100.0	8.6	8.6	0.0

Table 3: The coverage, effective constraints and percentage, error displacement, UAS, LAS, ΔUAS over the corrected arcs, and the constrained and cascaded Δ for MSTparser over WSJ 22 (covered by ZPar).

Error class	cover	eff	eff %	disp	UAS	LAS	ΔUAS	Δc	Δu
Baseline	100.0	-	-	-	91.7	89.2	-	-	-
NP attachment	95.6	277	4.3	4.8	94.9	92.7	2.4	1.0	1.4
NP internal	98.2	197	3.0	3.0	93.2	91.1	1.2	0.7	0.5
Modifier attachment	96.8	303	7.5	3.9	94.0	92.3	1.8	1.1	0.7
PP attachment	98.3	357	12.4	3.9	93.8	91.4	1.7	1.3	0.4
Coordination attachment	97.7	240	16.2	5.8	93.5	91.1	1.3	0.9	0.4
Clause attachment	96.7	166	13.0	5.6	93.4	91.2	1.2	0.6	0.6
Root attachment	99.1	57	4.3	9.9	92.4	89.9	0.5	0.2	0.3
Punctuation attachment	93.2	430	13.0	7.3	94.5	92.1	1.6	0.2	1.5
Other attachment	94.3	187	6.3	5.5	94.2	92.7	1.3	0.7	0.6
All attachments	98.5	2760	8.8	5.8	100.0	100.0	8.0	8.0	0.0

Table 4: The coverage, effective constraints and percentage, error displacement, UAS, LAS, ΔUAS over the baseline, and the constrained and cascaded Δ for ZPar over WSJ 22.

two new errors, with the parser incorrectly attaching the clausal complement headed by *expected* and the modifier headed by *about*. In fact, correcting the `ccomp` arc in isolation rather than the `root` arc leads to MSTparser producing the full correct analysis for the second half of the sentence (though again, it does not repair the separate noun phrase error). This example highlights why we have chosen to implement our evaluation as a set of constraints in the parser, rather than Kummerfeld et al. (2012)'s post-processing approach, as we cannot know that the parser will react as we expect it to when repairing errors.

Tables 3 and 4 summarise our results on MSTparser and ZPar, calculated over the sentences covered by ZPar in WSJ 22. Results over the full WSJ 22 for MSTparser were consistent with these figures. We focus on discussing UAS results in this paper, since LAS results are consistent.

The UAS of constrained arcs in each experiment is the expected 100%. Effective constraints repair an error in the baseline, and the effective constraint percentage is this figure expressed as a percentage, i.e. the *error rate*. Error displacement is the average number of words that effective constraints moved an attachment point. The overall ΔUAS improvement is divided into Δc, the constrained impact, and Δu, the cascaded impact.

It is important to note that a parser may make a substantial number of mistakes on a particular error class (large effective constraint percentage), but correcting those mistakes may have very little cascading impact (small Δc), limiting the overall ΔUAS improvement. Conversely, there may be a class with a small effective constraint percentage, but a large ΔUAS due to a large cascading impact from the corrections, or simply because the class contains more constraints.

6.1 Overall Parser Comparison

When applying all constraints, ZPar has a 8.8% effective constraint percentage compared to 9.3% for MSTparser. This is directly related to the UAS difference between the parsers. Aside from coordination, where the parsers made a nearly identical number of errors, ZPar is more accurate across the board. It makes substantially fewer mistakes on clause attachments, punctuation dependencies, and NP attachments, whilst maintaining a small advantage across all of the other categories.

The relative rank of the effective constraint percentage per error category is similar across the parsers, with PP attachment, punctuation, modifiers, and coordination recording the largest number of effective constraints, and thus the most errors. This illustrates that the behaviour of both parsers is very consistent, despite one considering every possible attachment point, and the other using a linear transition-based beam search. ZPar is able to make fewer mistakes across each error category, suggesting that the beam search pruning is maintaining more desired states than the graph-based parser is able to rank during its search.

ZPar's coverage is 98.5% when applying all constraints. However, as the number of constraints is reduced, coverage also drops. This seems counter-intuitive, but applying more constraints eliminates more undesired states, leaving more space in the beam for satisfying states. Reducing the number of constraints permits more states which do not yet violate a constraint, but only yield undesired states later.

Punctuation constraints have the largest impact on coverage, reducing it to 93.2%. NP attachments, clauses, and modifier attachments also incur substantial coverage reductions. This suggests that ZPar's performance will degrade substantially over the sentences which it cannot cover, as they must contain constructions which are dispreferred by the model and fall out of the beam. Constraints with the smallest effect on coverage include root attachments, which only occur once per sentence and are rarely incorrect, and NP internal and PP attachments. For the latter two, the small displacements suggest that alternate attachment points often lie within the same projective span.

6.2 Noun phrases

Applying NP attachment constraints causes a 4.4% drop in coverage for ZPar, and the effective constraint percentage is below 5% for both parsers. However, these constraints still result in the largest ΔUAS for both parsers, at 2.6% for MSTparser and 2.2% for ZPar. This reflects the prevalence of NP attachments in the corpus.

ΔUAS is split evenly between correcting constrained (1.4%) and cascaded arcs (1.2%) for MSTparser, while it skews towards cascaded arcs for ZPar (1.0% and 1.4%). Most error classes skew in the other direction, while repairing one NP attachment error typically repairs another non-NP attachment error.

For NP internal attachments, both parsers have a similar error rate, with 206 effective constraints for MSTparser and 197 for ZPar. Although this is the second largest class, applying these constraints gives the second smallest Δu for both parsers. This implies that determining NP internal structure is a strength, even with the more complex OntoNotes 4 NP structure. Δc is also small for both parsers, reinforcing the limited displacement and cascading impact of NP internal errors.

Despite fewer effective constraints (i.e. less errors to fix), ZPar exhibits more cascading repair than MSTparser using both NP and NP internal constraints. This will be a common theme through this evaluation: the transition-based ZPar is better at propagating effective constraints into cascaded impact than the graph-based MSTparser, even though ZPar almost always begins with fewer effective constraints due to its better baseline performance. One possibility to explain this is that the beam is actually pruning away other erroneous states, while the graph-based MSTparser must still consider all of them.

Table 5 summarises the error classes of corrected cascaded arcs for the two NP constraint types, which are closely related. NP attachment constraints directly identify the head of the NP as well as its correct attachment, providing strong cues for determining the internal structure. NP internal constraints implicitly identify the head of an NP. We can see that for both types of constraints, many of the cascaded corrections come from the other NP error class.

Table 5 also shows that, compared to MSTparser, ZPar repairs nearly twice as many NP internal and coordination errors when using NP attachment constraints, and vice versa when using NP internal constraints. This suggests that ZPar has more difficulty identifying the correct heads for

	NP attachment		NP internal	
Error class	MSTparser	ZPar	MSTparser	ZPar
NP attachment	-	-	45	69
NP internal	43	80	-	-
Modifier attachment	65	68	24	30
PP attachment	26	36	2	10
Coordination attachment	37	67	20	41
Clause attachment	59	65	1	1
Root attachment	24	21	2	1
Punctuation attachment	79	80	26	41
Other attachment	68	76	7	11
Total	401	493	127	204

Table 5: The number of unconstrained errors repaired per error class when enforcing NP attachment and NP internal constraints for MSTparser and ZPar over WSJ 22.

nominal coordination, and often chooses a word which should be a nominal modifier instead.

6.3 Coordination, Modifiers and PPs

These categories are large error classes for both parsers, with constraints leading to UAS improvements of 1.3 to 1.7%.

PPs and coordination have high effective constraint percentages relative to the other error classes for both parsers. However, they are also amongst the most isolated errors, with only 0.3% and 0.4% Δu for MSTparser and ZPar respectively. These errors also have minimal impact on ZPar's coverage. Both classes seem to have relatively contained attachment options within a limited projective span. The small error displacements reinforce this idea.

Modifiers are relatively isolated errors for MSTparser (0.5% Δu), but less so for ZPar (0.7% Δu). There are substantially more modifier constraints than PP or coordination, despite all yielding a similar UAS increase. This suggests that modifiers are actually relatively well analysed by both parsers, but there are so many of them that they form a large source of error.

6.4 Clause attachment

MSTparser performs substantially worse than ZPar on clause attachments, with an effective constraint percentage of 17.9% compared to 13.0%, and Δc of 0.9% compared with 0.6%. MSTparser's error rate is the worst of any error class on clause attachments, while it is second to coordination attachments for ZPar. Attaching clauses is very challenging for dependency parsers, particularly considering the small size of the class.

ZPar again achieves a slightly larger cascaded impact than MSTparser (0.6% to 0.5%), despite having far fewer effective constraints. This implies that the additional clause errors being made by MSTparser are largely self-contained, as they have not triggered a corresponding increase in Δu.

6.5 Root attachment

Both parsers make few root attachment errors, though MSTparser is less accurate than ZPar. However, root constraints provide the largest UAS improvement per number of constraints for both parsers. Root errors are also the most displaced of any error class, at 9.3 words for MSTparser and 9.9 for ZPar. When the root is incorrect, it is often very far from its correct location, and causes substantial cascading errors.

6.6 Punctuation

Despite ignoring punctuation dependencies in evaluation, applying punctuation constraints led to substantial UAS improvements. On MSTparser, Δu is 0.1% (due to some punctuation not being excluded from evaluation), but Δc is 1.7%. On ZPar, the equivalent metrics are 0.2% and 1.5%. Enforcing correct punctuation has a disproportionate impact on the remainder of the parse.

For both parsers, punctuation errors occur more frequently than any other error type, with 469 and 430 effective constraints respectively (though the majority of these corrected errors are on non-evaluated arcs). ZPar's coverage is worst of all when enforcing punctuation constraints, suggesting that the remaining uncovered sentences will

Error class	MSTparser	ZPar
NP attachment	75	51
NP internal	25	27
Modifier attachment	33	43
PP attachment	45	55
Coordination attachment	87	106
Clause attachment	66	48
Root attachment	59	27
Other attachment	65	69
Total	455	426

Table 6: The number of unconstrained errors repaired per error class when enforcing punctuation constraints for MSTparser and ZPar.

contain even more punctuation errors.

Incorrect punctuation heads are displaced from their correct locations by 7.4 words for MSTparser and 7.3 words for ZPar on average, second only to root attachments. Given that we are using projective parsers and a projective grammar, the large average displacement caused by errors indicates that punctuation affects and is in turn affected by the requirement for non-crossing arcs.

Table 6 summarises the error classes of the repaired cascaded arcs when punctuation constraints are applied. MSTparser has a more even distribution of repairs, while ZPar's repairs are concentrated in coordination attachment. This shows that MSTparser is relatively better at coordination as a proportion of its overall performance compared to ZPar. It also indicates that the majority of punctuation errors in both parsers (and especially ZPar) stem from incorrectly identified coordination markers such as commas.

Punctuation is commonly ignored in dependency parser evaluation (Yamada and Matsumoto, 2003; Buchholz and Marsi, 2006), and they are inconsistently treated across different grammars. Our results show that enforcing the correct punctuation attachments in a sentence has a substantial cascading impact, suggesting that punctuation errors are highly correlated with errors elsewhere in the analysis. Given the broad similarities between Stanford dependencies and other dependency schemes commonly used in parsing (Søgaard, 2013), we anticipate that the problems with roots and punctuation will carry across different treebanks and schemes.

Punctuation is often placed at phrasal boundaries and serves to split sentences into smaller sections within a projective parser. Graph-based and transition-based parsers, both of which use a limited local context to make parsing decisions, are equally prone to the cascading impact of erroneous punctuation. Removing the confounding presence of punctuation from parsing and treating attachment as a global post-process may help to alleviate these issues. Alternatively, more punctuation-specific features to account for its myriad roles in syntax could serve to improve performance.

7 Conclusion

We have developed a procedure to classify the importance of errors in dependency parsers without any assumptions on how the parser will respond to attachment repairs. Our approach constrains the parser to allow only correct arcs for certain tokens, whilst allowing it to otherwise form the parse that it thinks is best. Compared to Kummerfeld et al. (2012), we can observe exactly how the parser responds to the parse repairs, though at the cost of requiring modifications to the parser itself.

Our results show that noun phrases remain challenging for dependency parsers, both in choosing the correct head, and in determining the internal structure. Punctuation, despite being commonly ignored in parsers and evaluation, causes substantial cascading errors when misattached.

We are extending our work to other popular dependency parsers and non-projective parsing algorithms, and hope to develop features to improve and mitigate the cascading impact of punctuation attachment errors in parsing. Given that constituency parsers perform strongly when converted to dependencies (Cer et al., 2010; Petrov and McDonald, 2012), it would also be interesting to investigate how they perform on our metrics.

We implement a robust procedure to identify the cascading impact of dependency parser errors. Our results provide insights into which errors are most damaging in parsing, and will drive further improvements in parsing accuracy.

References

Mohit Bansal, Kevin Gimpel, and Karen Livescu. 2014. Tailoring continuous word representations for dependency parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 809–815. Baltimore, Maryland, USA.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X Shared Task on Multilingual Dependency Parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-06)*, pages 149–164.

Daniel Cer, Marie-Catherine de Marneffe, Dan Jurafsky, and Chris Manning. 2010. Parsing to Stanford Dependencies: Trade-offs between Speed and Accuracy. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC-10)*.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-Fine n-Best Parsing and MaxEnt Discriminative Reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL-05)*, pages 173–180.

Stephen Clark and James R. Curran. 2007. Formalism-Independent Parser Evaluation with CCG and DepBank. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL-07)*, pages 248–255. Prague, Czech Republic.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008. Stanford dependencies manual. Technical report, Stanford University.

Jason Eisner. 1996. Three New Probabilistic Models for Dependency Parsing: An Exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 340–345.

Jason Eisner and A. Noah Smith. 2005. Parsing with Soft and Hard Constraints on Dependency Length. In *Proceedings of the Ninth International Workshop on Parsing Technology (IWPT-05)*, pages 30–41.

Richard Johansson and Pierre Nugues. 2007. Extended Constituent-to-dependency Conversion for English. In *Proceedings of the 16th Nordic Conference of Computational Linguistics (NODALIDA-07)*, pages 105–112. Tartu, Estonia.

Jonathan K. Kummerfeld, David Hall, James R. Curran, and Dan Klein. 2012. Parser Showdown at the Wall Street Corral: An Empirical Investigation of Error Types in Parser Output. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL-12)*, pages 1048–1059.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Ryan McDonald and Joakim Nivre. 2011. Analyzing and Integrating Dependency Parsers. *Computational Linguistics*, 37(1):197–230.

Ryan McDonald and Fernando Pereira. 2006. Online Learning of Approximate Dependency Parsing Algorithms. In *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL-06)*, pages 81–88.

Dominick Ng and James R. Curran. 2012. Dependency Hashing for n-best CCG Parsing. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL-12)*, pages 497–505.

Joakim Nivre. 2009. Non-Projective Dependency Parsing in Expected Linear Time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP (ACL-09)*, pages 351–359.

Joakim Nivre, Yoav Goldberg, and Ryan McDonald. 2014. Constrained arc-eager dependency parsing. *Computational Linguistics*, 40(2):249–257.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-Based Dependency Parsing. In *HLT-NAACL 2004 Workshop: Eighth Conference on Computational Natural Language Learning (CoNLL-04)*, pages 49–56.

Slav Petrov and Ryan McDonald. 2012. Overview of the 2012 Shared Task on Parsing the Web. In *Notes of the First Workshop on the Syntactic Analysis of Non-Canonical Language (SANCL-12)*.

Adwait Ratnaparkhi. 1996. A Maximum Entropy Model for Part-of-Speech Tagging. In *Proceedings of the 1996 Conference on Empirical Methods in Natural Language Processing (EMNLP-96)*, pages 133–142.

Alexander Rush and Slav Petrov. 2012. Vine Pruning for Efficient Multi-Pass Dependency Parsing. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-12)*, pages 498–507.

Anders Søgaard. 2013. An Empirical Study of Differences between Conversion Schemes and Annotation Guidelines. In *Proceedings of the 2nd International Conference on Dependency Linguistics*, pages 298–307.

Ralph Weischedel, Eduard Hovy, Mitchell Marcus, Martha Palmer, Robert Belvin, Sameer Pradhan, Lance Ramshaw, and Nianwen Xue. 2011. OntoNotes: A Large Training Corpus for Enhanced Processing. In Joseph Olive, Caitlin Christianson, and John McCary, editors, *Handbook of Natural Language Processing and Machine Translation: DARPA Global Autonomous Language Exploitation*, pages 54–63. Springer.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical Dependency Analysis with Support Vector Machines. In *Proceedings of the 8th International Workshop of Parsing Technologies (IWPT-03)*, pages 196–206.

Yue Zhang, Byung-Gyu Ahn, Stephen Clark, Curt Van Wyk, James R. Curran, and Laura Rimell. 2010. Chart Pruning for Fast Lexicalised-Grammar Parsing. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING-10)*, pages 1471–1479.

Yue Zhang and Stephen Clark. 2011. Syntactic Processing Using the Generalized Perceptron and Beam Search. *Computational Linguistics*, 37(1):105–151.